THEOR 205.1 US - CAI

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE
APPLICATION FOR LETTERS PATENT

Title: **METHOD FOR DEVELOPING BUSINESS COMPONENTS**

Inventor: Charles PACLAT

EXPRESS MAIL

Mailing Label Number EL 829764718 US

Date of Deposit October 11, 2001

I hereby certify that this paper or fee is being deposited with the United States Postal Service "Express Mail Post Office to Addressee Service under 37 CFR 1.10 on the date indicated above and is addressed to the Assistant Commissioner for Patents, Washington, D.C. 20231

Fani Malikouzakis
(Typed or printed name of person mailing paper or fee)

Fani Malikouzakis
(Signature of person mailing paper or fee)

C. Andrew Im
Reg. No. 40,657
Fulbright & Jaworski L.L.P.
666 Fifth Avenue
New York, N.Y. 10103

METHOD FOR DEVELOPING BUSINESS COMPONENTS

Related Application

This application is continuation of U.S. Provisional Patent Application Serial No. 60/239,409 filed October 11, 2000, which is incorporated herein by reference in its entirety.

Background of the Invention

There are many different design methodologies. However, most of these focus on a finding a solution to a single instance of a program. In doing so, they miss the more generic patterns that can be made into scaleable and reusable components.

A component is a reusable software building block; a pre-built piece of encapsulated application code that can be combined with other components and with handwritten code to rapidly produce a custom application. Components execute within a construct called a container. A container provides an application context for one or more components and provides management and control services for the components. In practical terms, a container provides an operating system process or thread in which to execute the component. Client components normally execute within some type of visual container, such as a form, a compound document, or a Web page. Server components are non-visual and execute within a container that is provided by an application server, such as a Web server, or a database system.

A component model defines the basic architecture of a component, specifying the structure of its interfaces and the mechanisms by which it interacts with its container and with other components. The component model provides guidelines to create and implement components that can work together to form a larger application. Application builders can combine components from different developers or different vendors to construct an application.

Components come in a variety of shapes and sizes. A component can be very small, such as a simple GUI widget (e.g., a button), or it can implement a complex application service, such as an account management function.

In order to qualify as a component, the application code must provide a standard interface that enables other parts of the application to invoke its functions and to access and manipulate the data within the component. The structure of the interface is defined by the component model.

An application developer should be able to make full use of the component without requiring access to its source code. Components can be customized to suite the specific requirements of an application through a set of external property values. For example, the button component has a property that specifies the name that should appear on the button. The account management component has a property that specifies the location of the account database. Properties can be used to support powerful customization services. For example, the account management component might allow a user to add a special approval process for withdrawals over a certain dollar amount. One property would be used to indicate that special approval functions are enabled, a second property would identify the conditions that require special approvals, and a third property would indicate the name of the approval process component that should be called when the condition exists.

The JavaBeans component model defines a standard mechanism to develop portable, reusable Java technology development components, such as widgets or controls. A JavaBeans component (a Bean) is a specialized Java class that can be added to an application development project and then manipulated by the Java development tool. A Bean provides special hooks that allow a visual Java development tool to examine and customize the contents and behavior of the Bean without requiring access to the source code. Multiple Beans can be combined and interrelated to build Java applets or applications or to create new, more comprehensive, or specialized JavaBeans components.

The Enterprise JavaBeans components model logically extends the JavaBeans component model to support server components. Server components are reusable, prepackaged pieces of application functionality that are designed to run in an application server. They can be combined with other components to create customized application systems. Server components are similar to development components, but they are generally larger grained and more complete that development components. Enterprise JavaBeans components (enterprise beans) cannot be manipulated by a visual Java development tool in the same way that JavaBeans components can. Instead, they can be assembled and customized at deployment time using tools provided by an EJB-compliant Java application server.

The Enterprise JavaBeans architecture provides an integrated application framework that dramatically simplifies the process of developing enterprise-class application systems. An EJB server automatically manages a number of tricky

middleware services on behalf of the application components. EJB component-builders can concentrate on writing business logic rather than complex middleware. The results are that applications get developed more quickly and the code is of better quality.

Object and Summary of the Invention

Therefore, it is an object of the present invention to provide a system and method that overcomes the shortcomings of the prior art system.

The present invention relates to a process for developing an Enterprise JavaBean (EJB) component by analyzing a business domain to generate functional requirements that models the business domain. The functional requirements are transformed into an EJB component model, preferably using a UML drawing tool. The resulting EJB component is then built from the EJB component model that encompass the business functionality of the business domain. The present process enables the user/developer to research business problems or domain (i.e., business project) and transforms them into EJB components.

In accordance with an embodiment of the present invention, the aforesaid process provides a mechanism by which underlying and reusable patterns can be uncovered and turned into scaleable software components.

The delivery of high quality software components is an extremely overriding concern in today's market. Use of such reusable software components will result in a system with increased quality since these components are pre-fabricated (i.e., pre-implemented) and pre-tested (i.e., certified) components. The greater the reuse of a component, the greater will be the quality of the system using it since the errors must have been eliminated.

Increased quality results in decreased development costs. Use of pre-tested reusable software components decrease the cost incurred in performing various quality assurance activities to detect the errors in the design specifications and implementation.

Various other objects, advantages, and features of the present invention will become readily apparent from the ensuing detailed description, and the novel features will be particularly pointed out in the appended claims.

Brief Description of the Drawings

The following detailed description, given by way of example, and not intended to limit the present invention solely thereto, will best be understood in conjunction with the accompanying drawings in which:

Fig. 1 is a diagram illustrating the participation flow in accordance with an embodiment of the present invention;

Fig. 2 is a diagram illustrating the parallel development process in accordance with an embodiment of the present invention;

Fig. 3 is a sample UML;

Fig. 4 is an example of class and stereotype;

Fig. 5 is an example of inheritance;

Fig. 6 is an example of aggregation and multiplicity; and

Fig. 7 is a block diagram illustrating the implementation process or phase in accordance with an embodiment of the present invention.

Detailed Description of the Invention

The present invention provides a new component development (NCD) process for discovering and implementing business components. The NCD process enables the user or operator to research business problems and/or business models (i.e., business project), and turn them into Enterprise JavaBean™ (EJB™) components. The resulting components eventually encompass all of the common business functionality for a particular business domain.

EJB is a component architecture for creating scalable, multi-tier, distributed applications, and it makes possible the creation of dynamically-extensible application servers. EJB provides a framework for components that may be "plugged in" to a server, thereby extending that server's functionality. Enterprise JavaBeans™ (EJB™) technology defines a model for the development and deployment of reusable Java™ server components. Components are pre-developed pieces of application code that can be assembled into working application systems. Java technology currently has a component model called JavaBeans™, which supports reusable development components. The EJB architecture logically extends the JavaBeans component model to support server component.

Server components are application components that run in an application server. EJB technology is part of Sun's Enterprise Java platform, a robust Java technology environment that can support the rigorous demands of large-scale, distributed, mission-critical application systems. EJB technology supports application development based on a multitier, distributed object architecture in which most of an application's logic is moved from the client to the server. The application logic is partitioned into one or more business objects that are deployed in an application server.

The present invention divides the component family into phases of implementation, so that the products can be quickly developed and introduced to the market, thereby reducing the time to market. That is, the phased approach of the present invention involves two distinct types of NCD projects. The first type involves user's initial version or prototype, i.e., user's first foray into a given business domain, and the second can be characterized as a follow on to an existing component group that adds additional functionality, i.e., updates or enhancements.

The NCD process generates object oriented representations of the software components that facilitate a vast array of business solutions. It identifies the key steps that are needed to build these components and provides guidelines for the order of their execution. It is well-known in the art that developing business objects differs from developing end user applications. The traditional end user is the person that is using an application to complete their daily tasks. Whereas, in the case of business objects, the end users are software developers and business analysts who customize these components to provide business solutions that the customers demand and need.

Business objects development also differs from the development of traditional software tools and infrastructure. Generally, the end users of these traditional software tools are the software developers. Hence, such traditional tools rarely contain business domain specific logic. Thus, they can be designed and developed with no input from the business domain experts. However, business object development requires much greater collaboration between the business analysts and the software engineering teams to generate well-structured components that capture the most important facets of a business domain.

The ability to deliver EJB components in a timely manner is an overriding concern in today's extremely competitive marketplace. The up front research of the

NCD process enables the software developers and business analysts to define the segments of the business domain where the NCD process can add value to the project.

It is appreciated the steps of the NCD process are not executed in a rigidly defined sequence. The NCD process is an integration development process. The only hard and fast rule is that a market analysis is done up front, thereby allowing the design and implementation phases to be completed in an environment of full disclosure. Preferably, a very scaled back version of the components is delivered after the first iteration of the NCD process. Each iteration of the NCD process provides feed back to the analysis and design phase, thereby resulting in a product that is increasingly more complete, i.e., having more functionalities.

An aspect of the NCD Process is the overlapping of team members as development passes through the various phases as shown in Fig. 1. Each phase of development involves participants from the functional organizations that follow it in the process. The initiation of a new suite of components is within the product management team. It is important that there be significant contributions from the sales and professional services teams. Typical functional representation can be 50% Product Management, 20% Professional Services, 15% Sales, and 15% Engineering.

The analysis phase 110 is a more detailed analysis of the functional requirements from a standpoint of implementing the functionality of a set of components. Typical functional representation can be 60% Product Management and 40% Engineering 120.

In the design phase 120, the NCD process turns the functional requirements into an object oriented model that encapsulates the data model and the process model. At this point the weighting can shift to 75% Engineering and 25% Quality Assurance.

The implementation phase 130 consists of a more detailed version of the architectural design, implementation of the business logic, and software testing of the components. At this point the typical functional representation can be 70% Engineering, 20% Quality Assurance, and 10% Documentation.

The release and support phase 140 involves delivering the components in a timely manner into the market place. At this point the weighting can be 50% Quality Assurance, 30% Documentation, 20% Support, and 10% Engineering.

In accordance with an embodiment of the present invention, the analysis phase 110 analyzes the customer feedback to incorporate the changes in the newer versions

of the components. It is important to delegate the required upgrade based upon its type, i.e., future version, dot versions (2.1, 2.2, etc.) or bug fixes to the corresponding phase in the process. The new version might include not only new functionality, but also bug fixes to code they already use, performance improvements, support for new platforms etc. The typical weighting at this point is 60% Support, 20% Sales, and 20% Professional Services.

Preferably, the phases or steps of the NCD process can be performed in parallel to improve time to market as shown in Fig. 2. This is especially true with regard to the implementation of the components and the building of the demonstration or prototype application. It is expected that once the initial design of the components is complete, the implementation of the components and the building of the prototype will provide opportunities for improvement.

In general the development process will involve parallel iterations of the product and associated demonstration applications as shown in Fig. 2. The NCD process of the present invention enables user to quickly develop products based on the Enterprise Java Beans component technology or platform that are extensible and configurable.

Analysis Phase

In the analysis phase 110, a functional requirements document is generated which defines the scope of the business functionality for a new set of components. The functional requirements document is a single document that summarizes research into an entire business domain. The "divide and conquer" approach is taken such that once a wide reaching NCD is completed it is likely that it will be divided into additional sub domains that are then recursively refined as NCD's of their own. This process continues until manageable units of functionality are arrived at.

In accordance with an embodiment of the present invention, the functional requirements document includes a summary of a list of input and an eFunction Matrix, more fully described herein. The summary can include an overview of the business problem being addressed and the concerns of a typical end user of the targeted solution.

Once the scope of the business problem is defined, the process of identifying resources (i.e., list of inputs) that relate to the business problem, i.e., business domain in question, are identified, assembled and summarized. The list of inputs can include, but is

not limited to, interacting components, industry analysts, related industry standards, commercial packages, related engagements and system integrators, additional in-house resources, and eFunction matrix.

The interactive components are components or other parts of the system that a particular component may potentially interact with. It can be a simple bulleted list of items naming the component and what functionality is needed from that component. For example, such interactive components list can comprise: (1) interacts with Inventory component to know what is in stock; (2) interacts with the Shipping component to determine how much shipping will cost; and (3) might use the Payment component to authorize payments. The interactive component list is also useful for cross-NCD dependency tracking.

Since in many cases industry analysts have already researched a given sector or business domain, the first step towards gaining an understanding of the given business domain is to gather, read and analyze the targeted analysts reports. Another source of information are standards bodies and professional societies. These organizations often publish standards that represent the functionality common across an array of vendors. Preferably, the NCD process collects these publications and standards into a repository. Also, a list of member companies in such consortiums or organizations provides insights into various solutions available or offered by various vendors in a given business domain.

In addition, the NCD process gathers the reference documentation and feature sets of commercial software packages. For example, a superset of the most prominent features of the commercial software packages may provide a solution to the problem that is being addressed. The union of the features in these products generally represents the baseline functionality for a product offering. Accordingly, the NCD process may provide generic interfaces, i.e., adapters, to take advantage of third party solutions to one or more problems to efficiently and quickly develop business components.

Further, it is important to include representatives from the customer personnel, architectural and professional services teams in the analysis phase of the NCD process to develop new business components or extend existing business components to meet the needs of the customer. Accordingly, a corporate knowledge base of documentation from all professional services engagements should be maintained and

referenced. Additionally, the goals of the NCD process should be published to elicit inputs from various company personnel of the customer.

Upon completion of the list of inputs, the NCD process generates an eFunction matrix or table. The eFunction matrix advantageously provides a single place in which all of the options are quantified and can be compared and contrasted. Generally, the features or eFunctions are described in a summary of less than five words and collected into groups or packages. For example, an eFunction Matrix for "shopping component" is set forth in a following table:

Table 1: eFunction Matrix: Shopping							
Efunction		Development Status			Competitors	Customers	Partners
Package	Summary/Description	Status (0 to 10)	Importance (0 to 10)	Difficulty (0 to 10)			
CRM	Integration with voice, email and web CRM system		5				
CRM	Provide metrics on the business operations		5				
Fulfillment	Can pass data to the Order Fulfillment system		10				
Invoicing	Discount policies can be applied at customer level	0	7				
Invoicing	Discount policies can be applied at invoice level	0	7				
Invoicing	Misc. charges can be applied to invoice or packing list	0	7				
Invoicing	Discounts can be applied to items in cart		8				
Invoicing	Total of items in cart can be calculated		8				
Invoicing	Support for terms of sale	0					
Item	Items can have dimensions, weight	0	7				
Item	Items can have a tax code	0	9				
Item	Items can be added into shopping cart		10				
Item	Quantities of items in shopping cart can be updated		10				
Order	Multiple delivery methods	0	5				
Order	Allow the purchasing of only a portion of the shopping cart		5				
Order	Sending shipment as a gift	0	9				
Order	Customization of order cost calculation policy	10	9				

Table 1 (cont.): eFunction Matrix: Shopping							
Efunction		Development Status			Competitors	Customers	Partners
Package	Summary/Description	Status (0 to 10)	Importance (0 to 10)	Difficulty (0 to 10)			
Order	The system works for known users		10				
Order	The system works for anonymous users		10				
Order	Special instructions for delivery	0	6				
Order	Support for back order cancellation date	0	9				
Payments	Can manage multiple payment methods		5				
Payments	Can manage coupons and gift certificates	0	5				
Payments	Can pay with purchase orders		6				
Payments	Can authorize payment methods		7				
Payments	Can manage at least 1 payment method		8				
Payments	Can pay with credit cards		10				
Shipping	Exact shipping cost can be calculated		2				
Shipping	Approximate shipping cost can be calculated		5				
Shipping	Customization of shipping cost calculation policy	10	5				
Tax	Tax can be calculated on a line by line basis	0	9				
Tax	Interact with ad servers to cross-sell and up-sell		4				
Tax	Interact with Shopping/Advisor to recommend products		7				

In accordance with an embodiment of the present invention, the eFunction Matrix table is divided into five major sections. The first section lists each "eFunction" and the package or grouping to which it belongs. The second section lists the development status with regard to any pre-existing functionality (i.e., a range from 0 to 10, where 10 means the development is complete), the importance of the feature as a gauged by competitors, customers, and partners (i.e., a range from 0 to 10, where 10 is a high priority item or must have eFunction), and difficulty of extending pre-existing functionality or components to provide the required or specified eFunction (i.e., a range from 0 to 10, where 10 is the hardest). The competitors section contains a column for each software solution that can be considered a competitive product offering required functionality, i.e., eFunction. For each eFunction in the competitors section, an ad-hoc scoring from 1-10 is provided where 10 is a complete solution to that specified eFunction. The customers section records both customer requests and customer implemented or extension of the business components to encompass additional functions that were not originally provided for. The last or partners section records potential opportunities to leverage partners' products or partnerships. It is appreciated that in certain cases it may be difficult to distinguish partners from competitors until the analysis phase of NCD is completed.

Design Phase

In accordance with an embodiment of the present invention, the design phase 120 of the NCD process uses the unified modeling language (UML) modeling tool to perform the object oriented analysis and design. The UML is a general-purpose notational language for specifying and visualizing complex software, especially large, object-oriented projects, i.e., an industry standard notation for describing object oriented system. For example, the user can use the UML modeling tool to design the business components and the "Smart Generator" to generate the requisite java source code that implements the designed business components. The Smart Generator is described in assignee's co-pending U.S. Patent Application entitled "Smart Generator" filed October 10, 2001, which is incorporated herein by reference in its entirety.

The focus of the design phase 120 is to perform an in-depth analysis of the resources defined in the analysis phase 110. In the design phase 120, the NCD process refines the

general descriptions generated in the analysis phase 110 into a design document from which implementation or construction of the components can be started. That is, the NCD process turns the functional requirements into an object oriented model that encapsulates the data mode and the process model.

In the design phase 120, the NCD process refines the functional requirements document generated in the analysis phase such as filtering out non-interacting components to provide a list of interacting components, whose functionality will be needed by a specific component.

Also, the NCD process lists the individual users and or external systems that are to interact with the newly developed business component, such as end users, institutions providing a particular service, specific proprietary legacy system, and system accessed by industry standard protocols, etc. Preferably, the refined functional requirements document identifies each such listed entity and describes an entity's role in the business process.

The NCD process defines "use cases" in UML, which describe the business process in simple narrative form. The relationships between the actors and the use cases are visualized in use case diagrams and then these use cases are transformed into interaction diagrams that describe the operations that actors initiate on objects as well as object-to-object operations. This enables the developer/designer of the system to identify the components and the operations that will need to be performed upon them. In accordance with an embodiment of the present invention, a prototype is developed as part of the use case exercise. This prototype application can take the form of a graphical front end that implements the system from the actors perspective, thereby providing the designers/developers the opportunity to explore the use cases in depth.

The foundation package is a set of classes from which EJB components are built. These set of classes provide the building blocks for the value added features of the business components. Most of the classes that are generated from the model are derived from classes in the foundation package. For example, the "theory.smart.ebusiness" package contains classes that are built on the foundation package. To simplify the complexity of the UML diagrams, the foundation package relationships are described through class stereotypes rather than inheritance. Each of these class stereotypes is used

to model certain behaviors and implies the presence of additional methods. The following class stereotypes are used in the foundation package: Belongings, Sessions, Entity, Configurable Entity, Business Policy, Workflow, and Smart features.

Belongings

A Belonging, the simplest form of Smart Component, preferably, eBusiness Smart Component, is a lightweight, local object that can be serialized. A Belonging gets its name because it must "belong" to, or be acquired from, another object, typically a Session or Entity. It must be serializable so that it can be persisted with the class to which it belongs and passed remotely as a parameter.

One of the characteristics of a Belonging is that it must be implemented using the abstract factory pattern. This means that for each Belonging there is a home class, an interface, and at least one implementation of that interface. In other words, an EJB object is not created using new class, but instead created using a home class. Since access to the object is through an interface, there is a guaranteed level of abstraction. This provides a great deal of flexibility because it enables the developer/user to substitute implementations. For example, one can make the object remote without changing the code that uses it. Alternatively, one can substitute different business logic at runtime by changing the implementation returned by the home class.

Implementing all these classes by hand is lot of work. The component development process of the present invention has simplified the process by generating all of the necessary classes automatically, thereby enabling the developer/user to concentrate on modeling the attributes and methods so that they fit the desired business needs and requirements.

Sessions

Session components (implemented as Session EJBs) are used to model service-oriented objects. The key concept is that a Session is an object or Bean that provides access to a service implemented in itself or somewhere else on the network. Attributes of a session are used only to configure it for use during the lifecycle of that session. It is important to note that the attributes of a Session are not persistent. The business methods are the most important part of a Session.

Sessions provide a way of remotely implementing business logic, thus extending the reach of the client application. For example, when one needs to perform an extended set of operations on a collection of remote objects, it is desirable to create a "Manager". The Manager object can be co-located with the objects it will be operating on. This will advantageously reduce the network overhead and latency.

Sessions are also commonly used to provide an interface to a legacy system or to a service that is pinned to a specific piece of hardware. The remote interface allows the client software to access the remote device as if it were local.

Finally, by wrapping a subsystem and factoring out the functions common to similar systems it is possible to provide a level of redundancy, e.g., when there are multiple providers of credit card validation services. These systems would likely have similar function but different implementations. By creating a common interface to use the different implementations, it is possible to load balance between them or substitute one for the other.

Entity

An Entity (implemented as an Entity EJB) is an object with staying power. Persistence is the key aspect of an Entity Bean or object. An Entity object represents information persistently stored in a database. Entity objects are generally annotated with database transactions and can provide data access to multiple users. In its simplest forth, an instance of an entity could be the equivalent of a single row in a relational database. This is an over-simplification because each Entity may include collections of attributes and implement business methods.

Entities are representative of the attributes of which they are composed. This is what distinguishes them from Sessions, which represent a collection of services. As a general rule Entities do not implement sophisticated business logic, instead, they are the components that are acted upon.

There are two types of persistence in Entity Beans, Container-managed persistence and Bean-managed persistence. For Container-managed persistence, the EJB container is responsible for saving the Bean's state. Since it is container-managed, the implementation is independent of the data source. The container-managed fields need to be specified in the Deployment Descriptor and the container automatically handles the persistence. For Bean-

managed persistence the Entity Bean is directly responsible for saving its own state. The container does not need to generate any database calls. Hence, the implementation is less adaptable than the previous one as the persistence needs to be hard-coded into the Bean.

Configurable Entity

In addition to the standard qualities associated with an EJB Entity, the present invention provides dynamic configuration. Dynamic configuration is the ability to add properties and methods at runtime and is provided by the Configurable Entity. The Configurable interface allows the programmer to associate a named value with the Entity. These values are persisted separately so that they are permanently associated with the object without affecting the underlying schema.

When the value stored in a Configurable Entity is a method, the result is the ability to exchange the implementation of a method dynamically or a "Pluggable Method" which is the implementation of the "Strategy" pattern.

Business Policy

Configurable Entities can be arranged in a hierarchy of successors. When such hierarchy is in place, a request to retrieve a value from a Configurable Entity triggers an upward search through the hierarchy of successors until a matching value is found or the top of the hierarchy is reached. This is the implementation of the "Chain of Responsibility' design pattern.

The combination of "Pluggable Methods" and the hierarchy of succession referred to herein as the "Business Policy".

Workflow

For many business applications a simple mechanism to maintain internal state is all that is required to achieve a basic level of workflow. The present invention provides such a capability for defining and verifying the states and events that describe a business process. The developer can represent the business process as a state diagram and then verify the legitimacy of business method invocations with a single method call to ask for a transition.

Therefore, adding a step is as simple as adding a new state. The engine will then enforce the rule that this step must be taken without changes to existing code.

The components development process of the present invention has designed and integrated advanced features into the Smart EJB components, such as the SmartKey, SmartHandle, and SmartValue. Accordingly, these Smart features of the present invention considerably improve the ease of use and efficiency of the final system.

SmartKey

The EJB specification requires that for each Entity there is a class that represents the attributes of the primary key of that class. This Primary Key class is used to find and test the equality of instances of Entity objects. To accomplish these simple goals the EJB specification only requires that the Primary Key class must be serializable.

The SmartKey interface of the present invention extends this functionality and requires the implementation of the Comparable interface from the java collection API. This is so that SmartKeys can be easily compared and stored in ordered lists. The result is that it is easy to model relationships that require the ordering of Entities. The to "String" method of a SmartKey simplifies the implementation of profiling and debugging code.

SmartHandle

The EJB specification provides for the passing of lightweight references to Enterprise Java Beans through the use of Handles. A handle in EJB is an opaque type that can be converted to and from an EJB Object. A handle is required to implement a test for equality such that given two handles it is possible to determine if they refer to the same Session or Entity object.

For a Smart Entity component, a SmartHandle that includes the object's associated SmartKey can be generated. The SmartHandle excludes the capability of EJB Handle. Since the SmartKey implements the Comparable interface, it is a list of Smart Handles can be ordered without accessing the remote objects that they refer to. This simple mechanism greatly improves performance.

SmartValue

Each Entity is composed of the attributes that describe it. In order to encapsulate the remote objects all attributes must be read and written through accessor methods, typically named get <Attr > and set <Attr>. This has the negative consequence in that retrieving the attributes of an entity may result in many remote method invocations. To alleviate this problem the Smart features of the present invention provides a convenience class, derived from SmartValue, that contains a copy of all the top-level attributes.

Modeling Concepts

The Unified Modeling Language describes objects and their relationships graphically. The present invention has adopted this industry standard as a mechanism for simplifying the design and implementation of Enterprise Java Beans. Turning now to Figure 7, there is illustrated a sample UML for describing the UML notation that are used by the Smart Generator.

Classes and Stereotypes

Each of the rectangles in a UML diagram, as shown in Fig. 3, is a representation of a class in UML. There are generally three compartments in each class box. A compartment may be left out if it is empty or if the details of the contents are not pertinent to a particular diagram. The latter is often the case when an object from another package is being referred to.

The upper most box or compartment 810 is Fig. 4 holds the class name and its stereotype. A stereotype is a "sub-classification" of an element in the model. It is represented as the name of the stereotype enclosed in guillemets, as in «stereotype ». In the UML pretty much anything can be tagged with a stereotype. In Fig. 4, the Item class is stereotyped as a Configurable Entity. This means that it would have the qualities of one as described in the section Entity.

Attributes are listed in the second compartment 820. In UML, the name of the attribute is specified first followed by its type. The name and the type are separated by a colon. It is notable because it is different from the Java language. This works well for object oriented modeling which is generally an iterative process. Often tines a designer will list the attributes of class with out specifying types the first time through. The same technique holds true when specifying the arguments to a method. It is appreciated that the attributes can be

decorated with a stereotype. The stereotype precedes the attribute and is embedded in guillemets as before.

The third and final compartment 830 lists the methods. The return type is listed after the closing parentheses and is separated from the class definition with a colon. Often times the display of the parameters and the return value are suppressed on the UML diagram because they consume a great deal of space. When specifying attributes and methods in the UML, one can connote whether or not they are private, protected, or public. The "tilted brick" icon to the left has slight variations depending on the status of the attributes and methods.

An inheritance is depicted on a UML diagram as an unfilled arrow that points from the subclass towards its parent. In Fig. 5, the ItemPriceCalculationPolicy has a calculatePrice method through inheritance. The subclass shares all of the properties and attributes of its parent.

An aggregation is used to describe a containment relationship between classes. This is an alternative to simply defining an attribute with the type of the class. In UML, the contained object shares a life cycle with the containing object. That is, the containing object holds the only reference to it and is responsible for removing the object upon when it, itself, is removed.

An aggregation is depicted in UML with a line that extends from the containing to the contained item as shown in Fig. 6. The line begins with an oblong diamond that specifies a category of containment. A hollow diamond is used to show that the object is being contained by reference. A solid diamond specifies that the object is contained by value.

In accordance with an embodiment of the present invention, it a multiplicity for the object being contained can be also specified. Options are 1 (one to one), 0..1 (optionally null for references), or 0..* (one to many). As with all other elements of the UML it is possible to stereo type the relationship can be stereotyped. Preferably, an aggregation can be named as well.

Packages are used to group classes and other packages in to a hierarchy. Each package contains classes and/or other packages. When the classes of one package "use" the classes of another, this is depicted as a dotted line with an arrow in the appropriate directions, this same "use" notation can be applied to classes as well.

Beyond the standard mechanisms provided in the UML for defining components, other techniques designed to exploit existing message passing formats can be employed,

such as a document type definition for the eXtensible Markup Language (XML), can be employed. The NCD process can use the following guidelines to generate an initial component model using an existing message formats:

1. Objects should be identified from the message formats. For example, if there are "New Order" and "Cancel Order" messages in a given protocol it is advisable to create an "Order" object. In many cases, the existence of the telltale create, refresh, update, and delete messages signify that there is some underlying entity that needs to be modeled.

2. The methods on each object should be identified. This can be performed by simply translating a message such as "Cancel Order" to the "cancel" method on the "Order object". Alternatively, one can combine multiple messages into a single method call. For example, the messages "Begin Transaction", "End Transaction" and "Update Order" can be translated into an "update" method on the "Order" object that sends all three messages in the appropriate order.

3. The identity of the remote object is captured in a set of attributes that capture the primary key of the entity. Typically, the values will be named appropriately as "identifier", "name", or "key".

4. Determine session based components that simply specify that some action is to be performed but do not identify an entity.

Once the use case analysis is completed, the NCD process generated the initial component models using the UML class diagrams and the specialized stereotypes. The process of mapping UML to Enterprise Java Beans is described in assignee's co-pending U.S. Application entitled "Smart Generator", which is incorporated by reference it its entirety.

The test cases can then be generated to provide an important validation of the component design. The NCD process designs the test cases before implementation phase to incorporate any potential quality assurance issues into the design before the implementation of the component, i.e., implementation phase. This may involve incorporating comments or feedback from the quality assurance (QA) team. Performing this simple step early in the design phase by the NCD process advantageously lays the groundwork for developing high quality business components.

Preferably, the designed business components and related work products are reviewed for content and form by various individual, such as senior development staff from engineering, architecture, and professional services, as well as representatives from the sales and marketing teams. This provides an opportunity for the various members of the organization to provide additional feed back before implementation of the business component, thereby increasing the likelihood of acceptance by the end user.

Implementation Phase

The implementation phase 130 of the NCD is the implementation or building of the components themselves. In the implementation phase 130, the NCD process generates the relational mappings and deployment descriptors. This may include the definition of security roles and a set of unit tests to test or exercise the functionality. The implementation of the business component may be completed in parallel with a reference implementation.

The manner in which the NCD process implements or builds new components is described in conjunction with Fig. 7. During the implementation phase 130, NCD process generates the classes that represent the business components and their "Java Doc" from the component model, preferably the UML model in step 310. The components are then compiled and deployed with the simplest form of persistence. The completion of deployable components in step 330 makes it possible to begin the component implementation, unit test creation and end-user documentation in parallel.

In the implementation phase 130, NCD process installs the Bean class and its supporting classes in the EJB server with container-managed persistence (CMP) to complete or provide deployable components in step 330.

There are two types of persistence in Entity Beans, Container managed persistence (CMP) and Bean managed persistence (BMP). For Container managed persistence, the EJB container is responsible for saving the Bean's state. Since it is container-managed, the implementation is independent of the data source. The container-managed fields need to be specified in the Deployment Descriptor and the container automatically handles the persistence. For Bean managed persistence the Entity Bean is directly responsible for saving its own state. The container does not need to generate any

database calls. Hence, the implementation is less adaptable than the previous one as the persistence needs to be hard-coded into the Bean.

In the Container managed persistence, the EJB container must keep the object and the database synchronized i.e., the container fetches the data from the database, puts it in the Bean and writes the data back to the database (the ejbload() and ejbstore() methods is not used to access the database).

Also, during the implementation phase 130 of the NCD process generates documentation targeted for the end-user developer based on the design documentation and the java documentation in step 320. The NCD process eliminates references to any non-implemented functionality and product comparisons from the functional requirements document.

Further, the NCD process tackles quality assurance issues by designing tests that fully exercise the components, i.e., "black-box" testing before the components have been full implemented in step 340. That is, the unit tests are designed to fully test and exercise the implemented components in step 380.

Furthermore, in the implementation phase 130, the NCD process addresses the issue of legacy system integrations, i.e., the integration of legacy applications and data into a distributed computing environment in step 360. In many case, the business logic implementation of the legacy application is not object-oriented, but nevertheless it must appear to be object-oriented due to the proliferation and popularity of the legacy application in the distributed computing environment. Software components can be used as a "wrapper" to make these legacy applications appear as though they are object-oriented. Business logic may be added in the form of methods and the implementation of this business logic amounts to functional substance to the components.

Upon completion, implementation or construction of the component in step 350 and the end-user documentation in step 320, the NCD process synchronizes the two paths, i.e., verify the accuracy of the documentation. That is, the end-user document must coincide with the implemented component, any functionality changes to the component needs to be documented in step 370.

Also, the NCD process needs to test or verify the implemented component by conducting unit tests on the component in step 380. The NCD process completes the

implementation phase 130 by creating a reference implementation or prototype of the components and packaging of the deployable application in step 400.

The final step in the NCD process involves the creation of a demonstration application that fully utilizes the components in a functioning application (i.e., a completed application incorporating the implemented components) in step 350. The NCD process then verifies the accuracy and robustness of the component design in step 390. This can involve beta testing where the prototypes or pre-release version (i.e., a beta version) of components are distributed to the targeted audience or business population for their comments and feedback. Generally, a team of people/developer builds or implements the components and a different team of people assembles the demonstration application and/or final application to ensure that the documentation and designed components are both desirable and useful to user community. Any required changes to the design or documentation can be incorporated in an update.

Generally, the development of the demonstration application in step 350 can begin prior to the implementation of the component. In particular, the design of the user interface and the logical flow is based on the functional requirements document generated in the analysis phase 110 and as revised in design phase 120. It is appreciated that the reference implementation process should be initiated as early as possible to uncover any potential flaws in the design before the implementation phase 130 has proceeded too far.

Once the application/product (i.e., a newly developed EJB business component) development is completed and a reference implementation is created, the application/product can be released to end-users or customers. The release and support phase 140 involves various processes, such as the gathering of the documentation, updates to the release notes (as needed), creation of the installable image (e.g., an InstallShield image), final testing of the installable image, and distribution of the product (e.g., web release).

After the product (new developed EJB business components) is released to the customer, the NCD process supports the product to assist customer with any installation issues and any configuration or usage problems. Also, the NCD process can receive feedback from the customer via the customer support department, such as bugs, requests for new features and other comments and suggestions. The latest version of product

can not only include new functionality, but also bug fixes to code they already use, performance improvements, support for new platforms, etc.

In any product, managing backwards compatibility such that development progress is not halted and current customers are not angered by the constant changes is a difficult challenge. The challenges become greater as the software product includes more programmability – it is easier to change one's mouse motions to accommodate GUI changes than it is to reprogram all of one's Excel™ macros because the macro language just changed. EJB components, by their very nature, are used as integral portions of large and complex programs making managing backwards compatibility a great challenge. Accordingly, the sound management of the upgrades to the newly developed components is critical to the NCD process.

The bug fix only upgrade is characterized by the fact that it introduces no changes to the model or to the behavior of any method. It typically includes only bug fixes, although performance improvements, documentation changes and other modifications can be also included. It is appreciated that Belongings, being both server and client entities, cannot be upgraded at all. The user can install a Bug Fix upgrade without changing any code or doing anything to their Rose models, i.e., UML models. Users can just install the product on the server and developer machines and continue working as before. It is appreciated that some releases do not need to be installed on the developer machines. For example, performance improvements or fixes for bugs that only happen when the component is under heavy load are generally not required on the development machines.

In addition to allowing for bug fixes, the NCD process allows the user to add new classes as well as new methods to existing classes (i.e., add new behavior) in the release and support phase 140. A new method can be either a method with a new name, or a new overload for an existing method. It is appreciated that this type of upgrade does not delete an existing method or change the behavior of an existing method. Since this type of upgrade adds new behavior or changes a class definition, it is generally installed on all servers, clients and development machines.

A major release in the NCD process makes no commitment to backward compatibility. Accordingly, it needs to be installed everywhere and existing code needs to be ported to the new components.

It is appreciated that other upgrade types are also contemplated in the present invention. The three upgrade types discussed herein have close cousins, each with slightly differently semantics. For example, a Client Bug Fix upgrade is similar to a Bug Fix upgrade but it is installed on each client and developer machine. This allows, among other things, changes to Belongings. Another upgrade type is a New Component upgrade that allows bug fixes and the addition of new components but does not allow changes to existing ones.

Upgrading UML or Rose Models

When a user works with the Smart components in the UML drawing tool or Rose they do so by utilizing the *.CAT files (Rose Category files). By doing this, the model for the Smart components are stored in different files than the model for the user's components and classes. By updating the *.CAT files, the NCD process causes the UML drawing tool or Rose to add both new components as well as new behavior to existing components without requiring the user to make any changes to their model. Once the *.CAT files are updated, the next time the user opens a model utilizing the Smart components they will see the upgraded model.

The constraints on an Added Behavior upgrade are that the behavior of existing methods is kept the same and that no methods are deleted. Unfortunately, it is virtually impossible to ensure that the behavior of existing methods stays the same. According, the NCD process includes a tool to ensure that no methods are deleted. The tool can simply run javap on the old and new class definitions and compare the output. A bug fix release has the added constraint that it does not change any class definitions.

It is appreciated that a major release places a large burden on the customer because they must rework their model and the corresponding code. Because of this, customer may not be able to upgrade an entire large system all at once. Therefore, the NCD process provides support for the following partial upgrades: using package naming scheme.

a) A developer might begin work using a new major release of our components while still maintaining a release using an older release, preferably on the same machine.

b) If they are running multiple applications on their web server, time may force them to upgrade some applications but not others to the newest major release.

c) They may wish to upgrade some parts of a system to the new components but not the rest. That is, they may wish to have a single application that uses multiple versions of our components.

While the present invention has been particularly described with respect to the illustrated embodiment, it will be appreciated that various alterations, modifications and adaptations may be made based on the present disclosure, and are intended to be within the scope of the present invention. It is intended that the appended claims be interpreted as including the embodiments discussed above, those various alternatives, which have been described, and all equivalents thereto.